GOAL-Eclipse User Manual

Koen V. Hindriks, Wouter Pasman, and Vincent J. Koeman

February 5, 2015

Contents

1 Introduction **5**

2 Installing the GOAL Platform **7**
2.1 System Requirements . . . 7
2.2 Installation . . . 7
2.3 Uninstalling . . . 8
2.4 Upgrading . . . 8
2.5 Customizing . . . 9

3 Creating and Managing Multi-Agent Projects **11**
3.1 Creating a New Multi-Agent Project . . . 11
3.1.1 Adding an Agent File to a MAS Project . . . 12
3.1.2 Adding an Environment to a MAS Project . . . 13
3.1.3 Adding a Launch Policy . . . 13
3.1.4 Importing Knowledge and Module Files . . . 14
3.2 Managing an Existing Multi-Agent Project . . . 15
3.2.1 Renaming, Moving and Deleting a File . . . 16
3.2.2 Automatic Completion . . . 16

4 Running a Multi-Agent System **17**
4.1 Launching a Multi-Agent System . . . 17
4.2 Running a Multi-Agent System . . . 18
4.3 Terminating a MAS and/or Its Agents . . . 19
4.4 Run Modes of a MAS and its Agents . . . 19

5 Debugging a Multi-Agent System **21**
5.1 Introspecting Agents . . . 21
5.2 Querying a Mental State . . . 21
5.3 Executing an Action manually . . . 22
5.4 Debugging Output . . . 22
Consoles . . . 22
Action History . . . 23
Agent Console . . . 23
Customizing the Logging . . . 23
5.5 Setting Breakpoints . . . 23
5.5.1 Stepping . . . 24
5.5.2 Setting Code Line Breakpoints . . . 25
5.6 Prolog Level Debugging . . . 26
5.6.1 Prolog Exceptions . . . 26
5.7 Runtime Preference Settings . . . 27

6 Known Issues and Workarounds **29**
6.1 Reporting Issues 30

Chapter 1

Introduction

This user manual describes and explains how to:

- Install the Goal platform,
- Start and use the Goal plug-in for the Eclipse IDE, and
- Run or debug a MAS

This document does not introduce the agent programming language Goal itself. For this we refer the reader to the Goal Programming Guide [3] and Goal's website [4].

Chapter 2

Installing the GOAL Platform

This chapter explains how to install the GOAL platform. We first describe the system requirements of the platform. Please verify that your system meets the minimal system requirements.

2.1 System Requirements

The GOAL platform runs on Windows, Macintosh OSX and the Linux operating system. For information on the exact versions of these OSs, please check GOAL's website at http://ii.tudelft.nl/trac/goal.

The GOAL platform requires Java (SUN or OpenJDK) version 1.7 or higher. We recommend using Sun Java 1.7.

2.2 Installation

GOAL is available as a plug-in for **Eclipse Kepler** and up. Using **Eclipse Luna** is recommended. It can be installed as follows:

- *Install Eclipse*: If you do not have Eclipse installed yet, download the installation ZIP from https://www.eclipse.org/downloads, and extract it anywhere you like.
- *Start Eclipse*: If you have not done this yet, select a workspace directory. Note that special characters in the path of this directory are not recommended.
- *Add a link to the* GOAL *repository*: In the Eclipse menu, select 'Help → Install New Software → Add' to add a new repository (with any name you like) using the following URL: `https://raw.githubusercontent.com/goalhub/eclipse/master/dist`
- *Install the* GOAL *Plugin*: Select the 'GOAL Agent Programming' field (make sure 'Group items by category' is enabled), click Next and follow the installation steps. During installation, ignore (accept) any warnings about unsigned content. This might take a little while.
- *Restart Eclipse*: Restart when the installation is done (you will be prompted).

After having installed the plug-in, a new 'GOAL Perspective' is now available. Switch Eclipse to this perspective for the first time through 'Windows → Open Perspective → Other → GOAL'. A shortcut will be placed in the top-right corner. Furthermore, we recommend changing the following preferences in Eclipse ('Window → Preferences'):

- General → Workspace → Refresh using native hooks or polling → Enable
- Install/Update → Automatic Updates → Automatically find new updates and notify me → Enable
- Run/Debug → Console → Console buffer size (characters)→ Set to 999999 (the maximum)

Please note that if you are having performance issues whilst debugging, e.g. with many agents, the console buffer size can be reduced. The consoles for each agents and/or the action history can even be entirely disabled in the 'GOAL → Logging' preference category, where the logging itself can be fully tweaked as well.

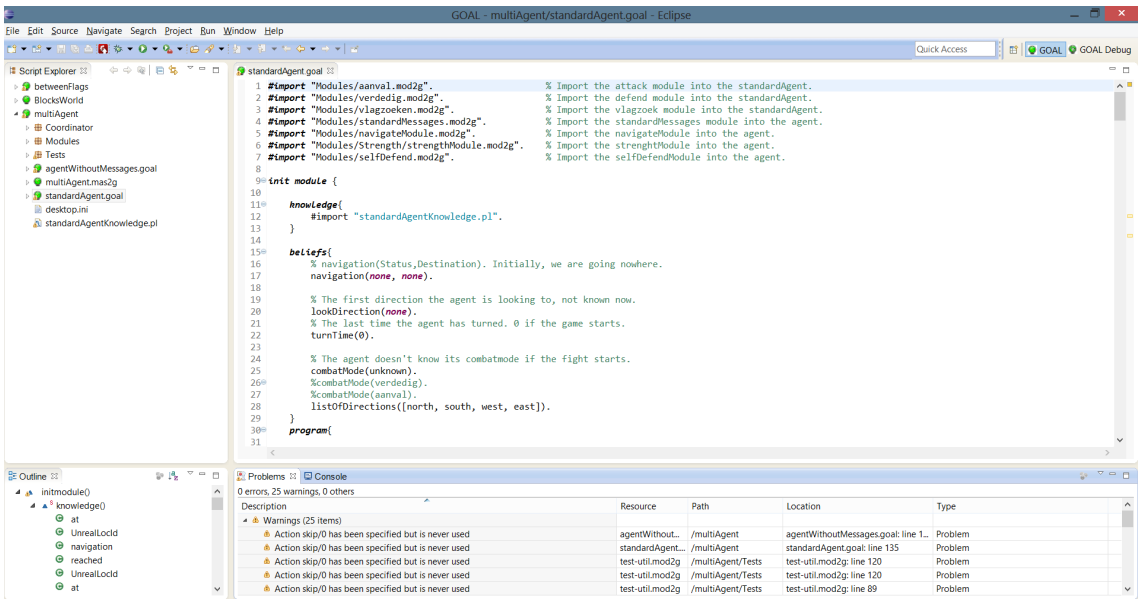

Figure 2.1: The GOAL perspective in Eclipse

2.3 Uninstalling

For uninstalling the GOAL plug-in, either remove your whole Eclipse installation, or follow these steps to uninstall just the plug-in:

- In Eclipse, select Help → About Eclipse → Installation Details.
- Select 'GOAL Eclipse' in the list of Installed Software, and click 'Uninstall'.
- Follow the remaining steps and restart Eclipse to complete the process.

2.4 Upgrading

Upgrading the GOAL plug-in to a newer version, when not using Eclipse's automatic update functionality, is possible in Eclipse by selecting: Help → Check for Updates, and following the

steps in the resulting dialog.

2.5 Customizing

Various GOAL platform features can be customized.

- Select Window → Preferences, and unfold the GOAL menu item.
- Select one of the three categories (Logging, Runtime, or Templates) to edit the related settings.

The default settings can always be restored using the 'Restore Default' button.

Chapter 3

Creating and Managing Multi-Agent Projects

This chapter explains how to create new, and manage existing multi-agent projects.

3.1 Creating a New Multi-Agent Project

How to create a new multi-agent project:

- In the GOAL perspective in Eclipse, select File → New → GOAL Project
- Enter a name, and optionally browse for an environment file to include in the project.
- Press Finish. A MAS file will be automatically created within the new project.

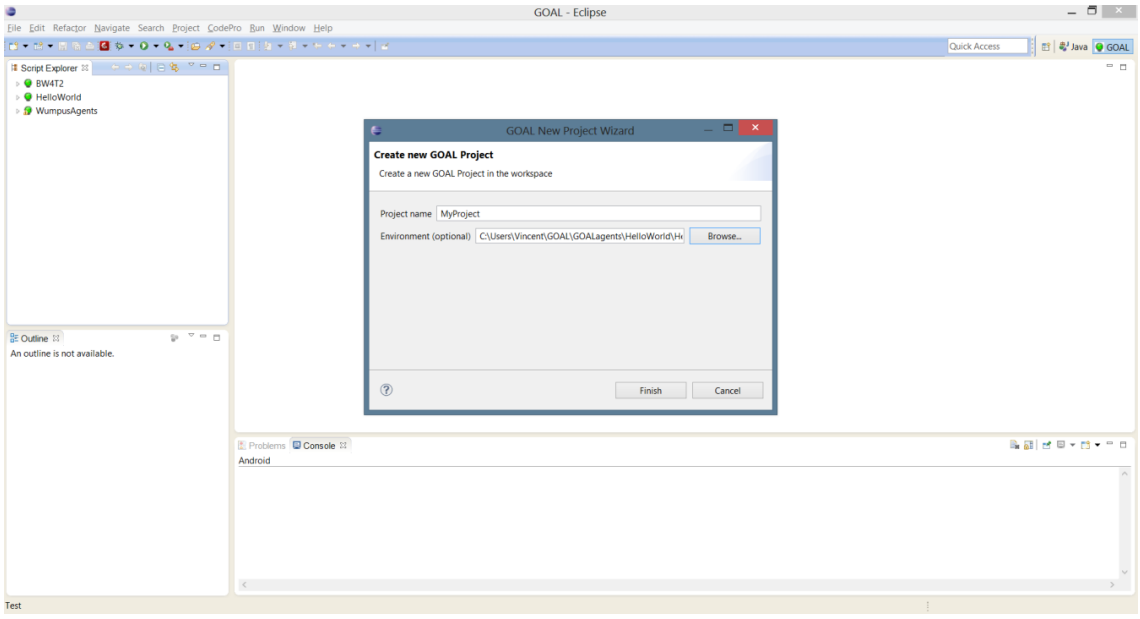

Figure 3.1: The New Project Dialogue Window.

MAS files are files with extension `.mas2g`. When a new MAS file is created, a **MAS template** is automatically loaded. This template inserts the the sections that are required in a MAS file; you should adapt the content of these sections of the MAS file to your needs. A MAS file must have an **agentfiles** and **launchpolicy** section; an **environment** section is optional. After opening a new MAS file, you should see something similar to Figure 3.2.

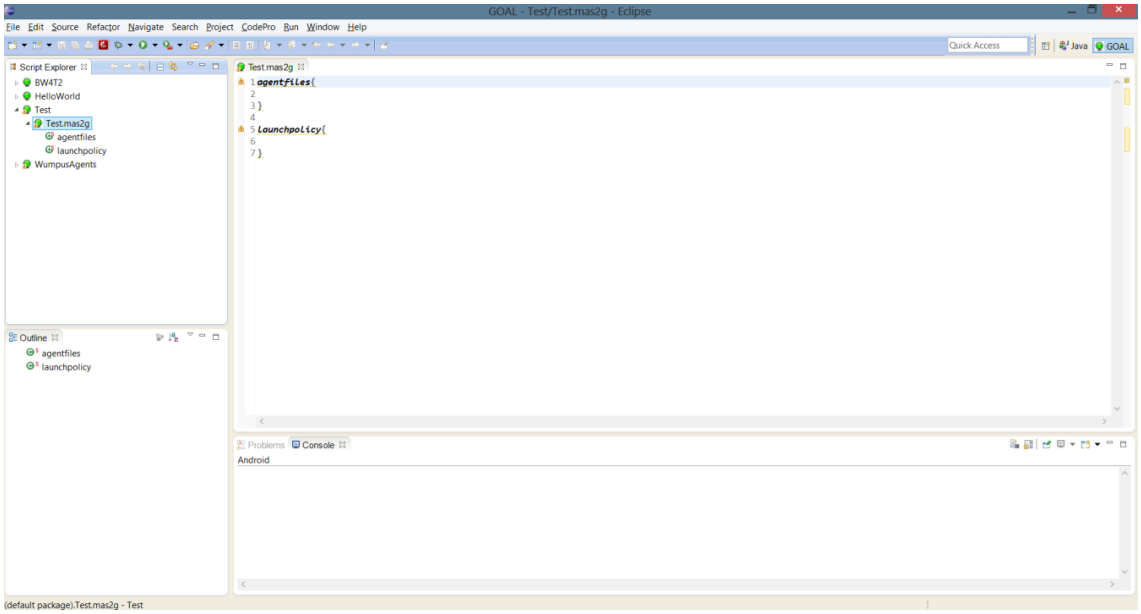

Figure 3.2: A new MAS File.

Warnings are displayed because the sections are empty. As can be seen, warnings and errors are shown at the line at which they occur, underlining the specific text causing the issue. In addition, the *Problems* tab can be used to list all errors/warnings. Any errors need to be fixed before we can launch a MAS. Note that multiple MAS files can be added to a single project, e.g. for different run configurations. Save a file either by using the Save icon in the menu, or by pressing CTRL-S (Windows and Linux) or APPLE-S (OSX).

3.1.1 Adding an Agent File to a MAS Project

How to add a new agent file to a MAS project:

- Right-click any file within the project or the project itself (in the Script Explorer), and select New → GOAL Agent File.
  Alternatively, if a file or project is already selected, File → New → GOAL Agent File can also be used.
- Give a (unique) name to the agent file, and press Finish.

Do not forget to add the agent file you have just created to the **agentfiles** section of a MAS file if required.

A new agent file is automatically loaded with a corresponding template. Again, warnings are displayed for a new file because the main module's program section is empty.

3.1.2 Adding an Environment to a MAS Project

An environment is added to a MAS project by first adding it to the project (which can be done for a new project or an existing one), and then editing the **environment** section in a MAS file. Note that the **environment** section is *optional* and that a multi-agent system can be run without an environment.

- Right-click any file within the project or the project itself (in the Script Explorer), and select Import.
  Alternatively, if a file or project is already selected, File → Import can also be used.
- Select General → File System, and press Next.
- Browse for the directory the environment file is in. In the right pane, the desired file(s) can now be selected.
- After having selected the right file(s), press Finish.
- Add `env = "filename"` to the **environment** section in the MAS file.
- If the environment supports initialization parameters (check the documentation that comes with the environment), add `"init = [<key=value>, ...]".` to the **environment** section in the MAS file; here, where `key` is the name of the parameter and `value` the value you choose to initialize the environment with.

Please note that you can also manually place the JAR file within the actual project location on your hard-drive; Eclipse will automatically update your project (or right-click the project and select Refresh to force a reload). References to environments in a MAS file are resolved by checking whether the environment can be found relative to the folder where the MAS file is located, or else, an absolute path should be specified to the environment interface file.

The GOAL platform supports the Environment Interface Standard (EIS) to connect to environments [2, 1]. Any environment that is compliant with the most up to date version of EIS that is supported can be used in combination with GOAL.

In order to actually run a MAS, it may also be necessary to initialize an environment by means of the environment's user interface. In the UNREAL TOURNAMENT environment, for example, you can create new bots that can be controlled by agents. Please consult the environment documentation for details. Depending on the configuration and launch rules of your MAS file, new agents are automatically created when a new bot is added. Refer to the Programming Guide for more information on this [3]. Note that environments often also have entities that cannot be controlled by agents, such as people taking the elevator in the Elevator environment or UNREAL bots that are controlled by the UNREAL engine itself.

3.1.3 Adding a Launch Policy

In order to create agents when a multi-agent system is launched, a launch policy for creating the agents is needed. A launch policy is a set of rules which specify when to launch an agent.

How to add a launch policy:

- Either, add a *launch instruction* of the form `launch <agentname> : <agentfile>.`
- Or, add a *launch rule* `when entity@env do launch <agentname> : <agentfile>.`

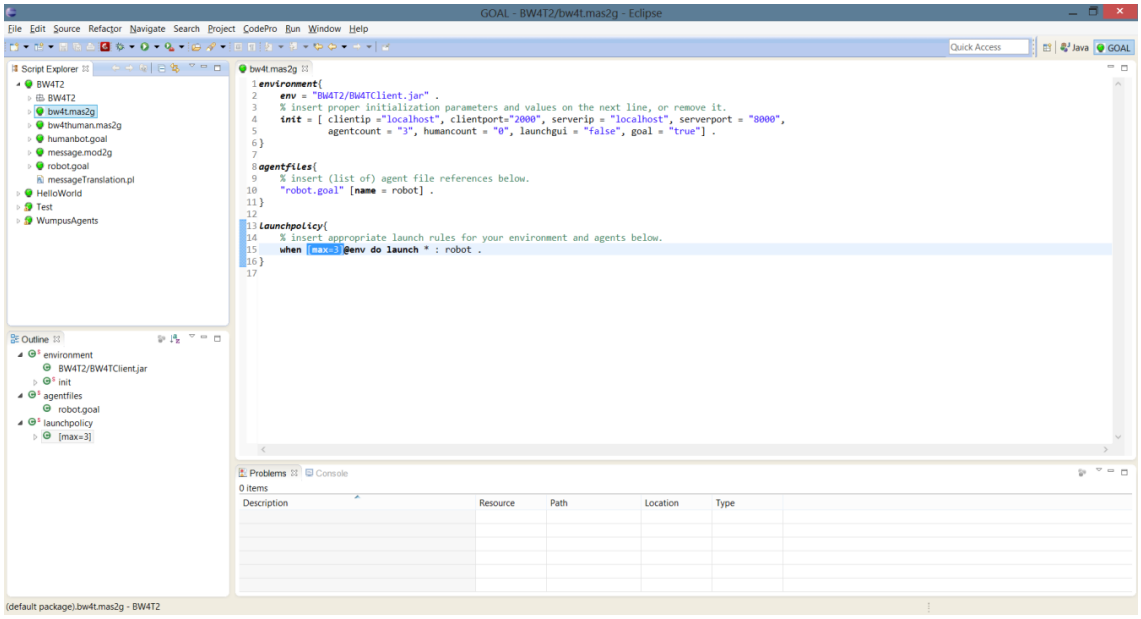

Figure 3.3: Launch Policy of the BW4T example project.

A launch instruction creates and adds a new agent to the MAS when the MAS is launched. A launch rule only creates an agent when a controllable entity in an environment becomes available (i.e., when the GOAL platform is notified that an entity has been born).

Agent name and agent file name can but do not have to be different.

Tip By using an asterix `*` instead of an agent name in a launch rule, the name of the agent that will be created will be determined by the name of the environment entity that the agent is connected to. This makes it easier to relate agent and entity in an environment. For example, an agent may be called `bot` if the entity in the environment is called `bot`. It is also possible to leave out the agent's name entirely, i.e. only referencing the agent file. In this case, the agent's name will be set to that of the corresponding file.

Finally, more than one agent can be connected to a single entity. The GOAL Programming Guide [3] provides a comprehensive explanation of launch policies.

3.1.4 Importing Knowledge and Module Files

When you start writing larger agent programs it is often useful to add more structure to the MAS project. GOAL provides the `#import` command to import contents from other files into an agent program. An important reason for moving code to separate files and importing those files is to facilitate **reuse**.

How to create and import a knowledge file into an agent file:

- Use the File → New → Prolog File menu to create a new `<knowledge>.pl` file, where `<knowledge>` is a name for the knowledge file.
- Select and cut all code in the **knowledge** section of the agent program.
- Paste the selected code from the **knowledge** section into the `<knowledge>` file, and save.
- Insert `#import "<knowledge>.pl".` in the **knowledge** section of the agent program, and save.

A procedure very similar to that for knowledge files can also be used to move complete modules to separate *module files* with extension `.mod2g`.

How to create and import a module file into an agent file:

- Use the File → New → GOAL Module File menu to create a new `<modulefile>.mod2g` file, where `<modulefile>` is a name for the module file.
- Select and cut all code for a module in the agent program.
- Paste the selected module code into the `<modulefile>`, and save.
- Insert `#import "<modulefile>.mod2g".` in the agent program at the top-level[a], and save.

[a] Top-level simply means that the code is not included in any other structure such as a module or other program section.

These are the two types of files that can be imported: Files that contain modules and files that contain knowledge. Even if an agent is not a very large program, it sometimes is useful to put some of the code in a separate module or knowledge files to facilitate *reuse*. The code in a **knowledge** section of an agent, for example, can be (re)used in another agent program. Reusing the same code by creating a separate file that is used in multiple agent files prevents possible problems introduced by modifying the same code that appears at different places in only one location. By using a dedicated file the code is needs to be maintained and modified in only one place.

3.2 Managing an Existing Multi-Agent Project

Loading an existing multi-agent system project can be done by means of the following steps:

How to import an existing project into Eclipse:

- Choose File → Import, select Existing GOAL Project (in the 'GOAL Agent Programming' category), and press Next.
- Browse for the `.mas2g` file that indicates the system that is to be imported.
- Optionally, check the Copy into workspace checkbox in order to make a copy of all the files, instead of using them directly in the new project.
- Press Finish. A new project with the name of the MAS file will have been created.

Please note that only files that are in the (sub)directory of the selected `.mas2g` file will be copied (or directly used) in the new project. Any external files will have to be moved into the project manually, by using the file system or the steps as explained in Section 3.1.2.

3.2.1 Renaming, Moving and Deleting a File

The name of a file can be changed by right-clicking on it, and selecting Refactor → Rename. Note that a file can also be moved through the Refactor menu. Deleting a file can simply be done by right-clicking on it and selecting Delete. A **warning** is in place, however: *Changing a file name or location that is part of a MAS project does not modify the contents of the file itself.* When changing an agent file name, or moving or deleting it, you need to manually change the **agentfiles** section of the MAS file. All of these operations can also be done on the file system itself.

3.2.2 Automatic Completion

Automatic code completion is fully supported.

Using auto-completion:

- After typing some initial code, it can be automatically completed by pressing CTRL-Space.
- If there is more than one option for completion, select an option from the pop-up menu (otherwise the word is completed right away).

Chapter 4

Running a Multi-Agent System

This chapter explains how to run a MAS project. Running a multi-agent system means launching or connecting to the environment the agents perform their actions in (if any), and launching the agents when appropriate. As explained in Section 3.1.3, which environment is launched depends on the **environment** section of the MAS file that is being run. Which agents are launched depends on the launch policy specified in the same MAS file.

4.1 Launching a Multi-Agent System

Launching a MAS means:

- Launching an environment referenced in the MAS file, if any, and
- Creating agents in line with the launch policy of the MAS.

How to launch a MAS:

- Right-click a MAS file in the Script Explorer.
- Select 'Debug As → GOAL'.

'Debug As' will automatically switch to the GOAL Debug Perspective, and place a shortcut to this perspective in the top-right corner (if it was not there yet). If you use the GOAL Debug Perspective for the first time, you will be asked if you want to open it. You should select 'Remember my decision' and 'Yes' in this case in order to debug properly. In the debug perspective, the **Debug panel** is shown at the top, as illustrated in Figure 4.1. This panel shows the processes that have been created by launching the MAS. In the Blocks World example, an environment process is started labeled *Blocks World*, as well as two agent processes named *stackbuilder* and *tableagent*. Also, the Blocks World GUI shown in Figure 4.2 should appear. All processes that are created are *paused* initially, which is indicated by the label just after the process names.

Before launching a MAS, make sure to check that the MAS project is clean. That is, make sure no warnings or errors have been produced. When you launch a multi-agent system, GOAL will not start the system when one or more program files contains an error. Apart from the need to remove parsing errors, it is also better to clean up program files and make sure that no warnings are present.

Please note that it also possible to run a MAS without a visual interface by selecting 'Run As → GOAL' instead of 'Debug As → GOAL'. The Run functionality will run the MAS through a console (at the bottom of the screen). At the top-right corner of the console, actions like terminating the run are available. You can search in any console as well ('Right click → Find/Replace' or 'Left click → CTRL+F'). This is useful for quick, light-weight testing of desired functionalities.

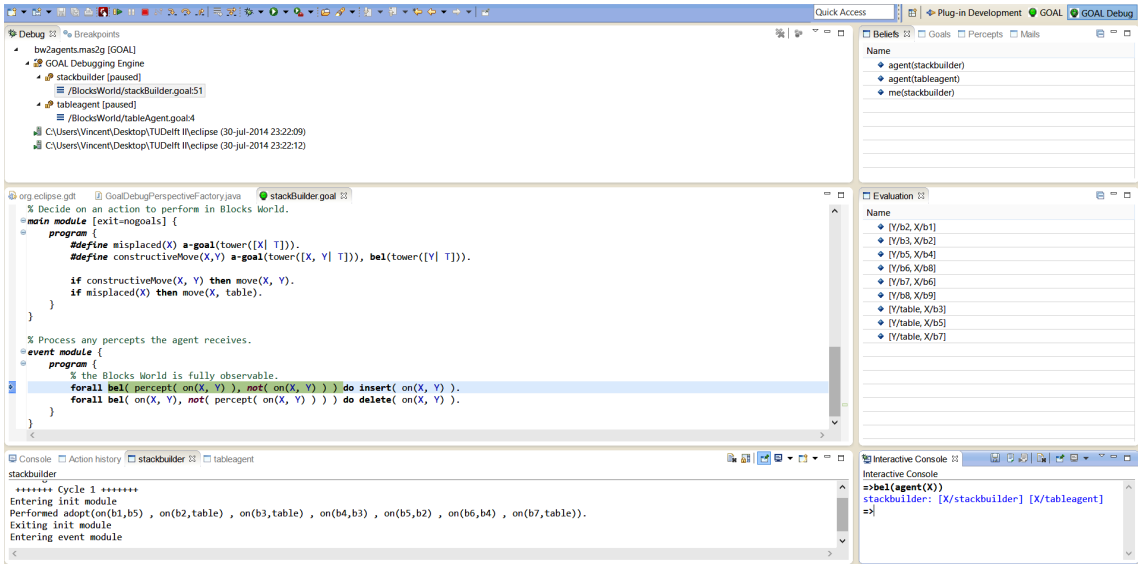

Figure 4.1: The GOAL Debug Perspective

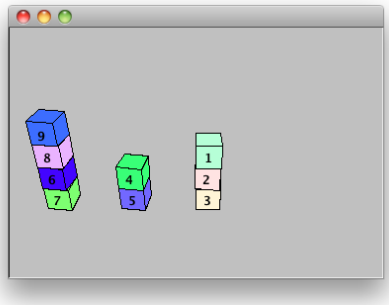

Figure 4.2: The Blocks World GUI after launching `bw2agents.mas2g`.

4.2 Running a Multi-Agent System

After an environment has been initialized, a multi-agent system can be run.

How to run a MAS:

- Select the top node in the Debug panel.
- Press the Run button in the tool bar.

All processes, including the environment and the agents, must be in *running mode* to be able to run the *complete* multi-agent system. In running mode, an agent will execute a complete reasoning cycle without pausing.

If the agents in a MAS are paused but the environment is not, the environment still may be changing due to processes that are not controlled by the MAS or simply because actions take time (durative actions). Vice versa, if an environment is paused but one or more agents in the MAS are

running, typically you will see complaints that the agent was not able to retrieve percepts from the environment.

4.3 Terminating a MAS and/or Its Agents

How to terminate a MAS:

- Select the top node in the Debug panel.
- Press the Kill button in the tool bar.
  Alternatively, press the red stop button of Eclipse's default console.

Terminating a MAS terminates the environment and all agents. An agent will also be *automatically killed* if the entity that it is controlling in an environment has been terminated. An agent that has terminated itself is not performing any actions any more. If a terminated agent is put back in running mode, the agent is restarted and initialized using the initial mental state specified in the agent program.

4.4 Run Modes of a MAS and its Agents

A MAS or agent can be either in the running, stepping, paused, or killed mode. Controlling the run mode of a process is done with the Run, Step (into/over/out), Pause, and Kill buttons in the tool bar. You can *control all processes* that are part of the MAS at once by selecting the top process node and using the relevant buttons. This is particularly useful when starting, stepping or killing all agents and/or the environment. Holding shift or ctrl whilst clicking on the nodes will allow you to hand-pick multiple processes.

Note It may happen that an agent tries to perform an action in an environment when the environment has been paused because the agent process and environment run asynchronously. In that case, the environment may throw an exception which will be printed as a warning in the console tab. The agent will proceed as if the action has been successfully executed.
Finally, note that environments can be in running and paused mode, but not in stepping mode.

Chapter 5

Debugging a Multi-Agent System

The GOAL runtime provides various options for debugging. While running a mas, GOAL provides a range of options for debugging which include the following:

1. options for viewing and controlling **debug output** produced while running a mas,
2. setting general and specific **breakpoints** to pause an agent on,
3. agent **introspectors** for inspecting the mental state of an agent, and percepts and messages received at runtime,
4. a code stepping mechanism.

5.1 Introspecting Agents

A useful tool to trace what your agents are up to is the *introspector*. An introspector allows you to inspect the mental state of an agent. In the debug perspective, the mental state of the currently selected agent is automatically shown at the right top. Note that this is only possible when an agent is paused or stepping.

An agent introspector consists of various tabs that allow you to access the current contents of the agent's beliefs, goals, percepts, and mails, percepts. This information is alphabetically sorted. Searching is possible in all of these views ('Right click → Find' or 'Left click → CTRL+F').

5.2 Querying a Mental State

Apart from being able to view the contents of an agent's mental state when it is paused, the debug perspective also allows you to pose queries for evaluation on the current mental state of the agent in Interactive Console at the right bottom.

How to evaluate a mental state condition:

- In the Debug panel, select the agent to query.
- Enter a mental state condition in the Interactive Console.
- Press Enter.

All solutions for the query performed are shown in the interactive console as a list of substitutions for variables. Consult the GOAL Programming Guide [3] for a detailed explanation of mental state conditions.

It is also possible to continuously evaluate and thus inspect one or more mental state queries whilst a system is running by using so-called 'Watch Expressions'.

How to continuously evaluate mental state conditions:

- Open the Expressions panel
- Click on 'Add new expression' and enter a mental state condition.
- Press Enter.

The evaluation of this expression will be shown for all agents, updated each second.

5.3 Executing an Action manually

You can also execute actions from the same Interactive Console. The actions that can be executed include (i) mental state actions which change the mental state of the agent (**adopt**, **drop**, **insert**, **delete**) and (ii) user specified environment actions that are executed in the environment. You cannot execute other actions such as combined actions, sending messages, or modules. Any input in the interactive console is automatically recognized as either an action or a query.

How to execute an action manually:

- In the Debug panel, select the agent that should perform the action.
- Enter the action in the Interactive Console.
- Press Enter.

Make sure that the action is closed, i.e., has no free variables. To execute an environment action, make sure that the environment is running. You may also need to figure out whether the agent that should execute the action is allowed to perform an action at the time you want to perform the action, e.g., it is the turn of that agent.

Note that an environment action is sent to the environment without checking its preconditions. The post-condition of the action also is not processed nor are any percepts related to the action passed to the agent. Finally, no feedback is provided whether an action was successfully performed or not.

5.4 Debugging Output

In debug mode, the bottom area contains various tabs for inspecting agents and the actions they perform, see Figure 4.1. Besides the main console tab, an action history tab is present that provides an overview of actions that have been performed by all (running) agents. In addition, when an agent is launched a dedicated console is added for that agent for inspecting various aspects of the agent program during runtime.

Consoles

The main console shows all important messages, warnings and errors that may occur. These messages include those generated by GOAL at runtime, but possibly also messages produced by environments and other components. When something goes wrong, it is always useful to inspect the console tab for any new messages.

Action History

The action history tab shows the actions that have been performed by running agents that are part of the multi-agent system. These actions include both user-defined actions that are sent to the environment as well built-in actions provided by GOAL.

Apart from an overview of what has been done by agents, the action history can also be used as a *simple means for debugging*. It can be used, for example, to check whether a particular action added for debugging has been performed. The idea is that you can add a dummy action with a particular content message to a rule and verify in the action log whether that action has been performed. To do so, include an action in the action specification of your agent named e.g. `debug(<yourMessage>)` with `true` as pre- and post-condition to ensure the action always can be performed and will not change the state of the agent but still will be printed in the action log output.

Agent Console

Upon launching an agent, a dedicated tab is added for that agent. This tab typically contains more detailed information about your agent.

The exact contents of the tab can be customized through the settings. Various items that are part of the reasoning cycle of an agent can be selected for viewing. If checked, related reports will be produced in the debug tracer of an agent. A more detailed explanation of these settings can be found in the next section.

Customizing the Logging

The logging behavior can be modified from the preferences in Eclipse. Access them through opening the GOAL category in Window → Preferences, and selecting the Logging page.

A timestamp with millisecond accuracy can be added to each printed log by checking the 'Show log time' checkbox. The action history and dedicated logging tabs for agents can also be turned on or off here.

The 'write logs to file' checkbox determines if GOAL writes all logs to separate files. The files will be stored in a directory within the current ly executed project. Logs for each agent, the action log, warnings, results from GOAL log actions etc are all written to separate files.

Most logs are in XML format, and each log entry contains a timestamp with millisecond accuracy. Debug messages for each ¡agent¿ are written to <agent>.log. All actions are written to historyOfActions.log. General info is written to goal.tools.logging.InfoLog.log. Warnings are written to goal.tools.errorhandling.Warning.log. System bug reports are written to goal.tools.errorhandling.SystemBug.log. Result of log actions are written to <agent>_<timestamp>.txt files. Be careful with this option, as many megabytes per second can be written. A new run generates new log files (made unique by adding a number to the names). By selecting 'Overwrite old log files', a new run uses the same file names, thus overwriting previously generated files.

The type of debug messages that are logged can be determined with the checkboxes in the Logging Options section.

In the Warnings section, you can turn on stack traces and/or detailed Java messages. You can also set the maximum number of times the same message is printed, to avoid flooding of the console.

5.5 Setting Breakpoints

A useful way to trace what your agents are doing is to step the multi-agent system and check how and which action rules are evaluated, which actions are performed, which percepts are received, etc. Stepping assumes natural breakpoints to stop an agent at. For GOAL agents, a number of such breakpoints have been pre-defined. These built-in breakpoints facilitate stepping and tracing of what happens when an agent is executed.

A second, alternative method to set breakpoints is by explicitly adding them to code lines in agent program files. A user can set breakpoints at various places in a program where the agent should stop when such a breakpoint is hit.

5.5.1 Stepping

The first method to step an agent and halt at breakpoints is based on a set of built-in global breakpoints that relate to the reasoning cycle of an agent. A list of these breakpoints is available in Eclipse by opening the GOAL category in Window → Preferences, and selecting the Runtime page. This preference page is illustrated in Figure 5.1.

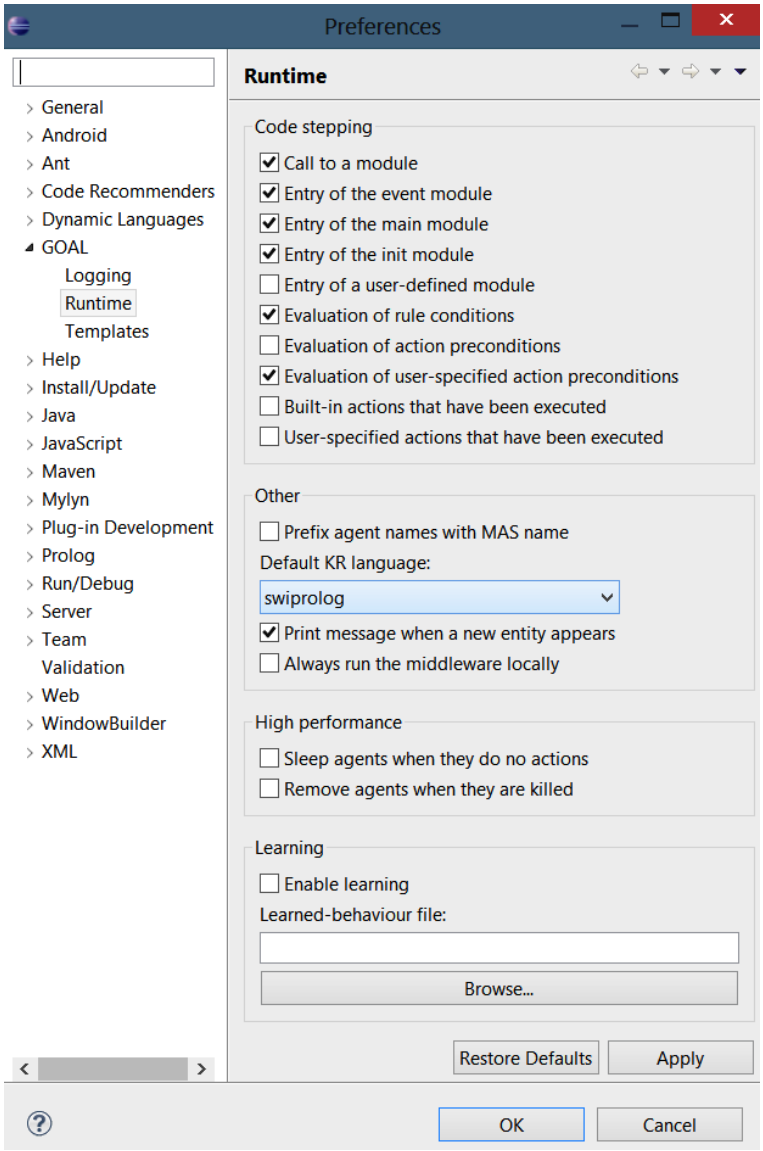

Figure 5.1: Setting the Code Stepping points

For each breakpoint, a checkbox is available. A checked box indicates that the stepping process will halt on the related code point.

Code stepping is supported for agents that are paused. Three different stepping actions are

available:

- *Step Into (F5)*: Step to the next evaluation point. By default, you will step to the next rule, action, or module that will be executed.
- *Step Over (F6)*: Jumps to the next module or rule and does not show any intermediate steps (of course, all code is evaluated as usual at all times). For example, if the event module is selected, selecting stepping over will jump immediately to the main module without showing any intermediate (evaluation) steps that are performed in the event module. Note in particular that stepping over the main module will have the agent perform a whole cycle; stepping therefore returns again to the main module. Check the agent's mental state to see what has changed.
- *Step Return (F7)*: Skips the remainder of the module the agent is currently executing. Of course, this only influences the stepping process itself. Also note that doing this whilst in the main module will make an agent complete its current cycles.

Code stepping inside a module depends on the rule evaluation order that is used in that module. For example, in the event module that by default uses the linearall order, stepping into will step from one rule to the next rule and exit the module only after having finished the evaluation of the last rule. In the main module that by default uses the linear order, stepping into will step from one rule to the next until a rule that is applicable has been found; after applying this rule the module is left. In a module that uses the random order, a rule that is applicable is randomly selected; code stepping thus does not need to start with the first rule! Finally, also note that a module may be re-entered immediately again if an exit option different from 'always' has been set.

In the window centered to the left below the overview of a MAS, the code of the agent that is being stepped is shown. If the agent is paused, the code line that the agent is at is highlighted. Stepping will proceed from that point on. Please note that editing files is not possible during debugging, but breakpoints can be changed at any time. Below the overview of the agent's bases, the current evaluation is shown when an agent is paused. What is shown depends on the current agent's code position and provides information related to code stepping as discussed above. More concretely, the window shows for which variable instances a rule condition holds (if any), or it shows whether a precondition of the selected action holds or not. Using 'Right click → Find' or 'Left click → CTRL+F' this view can also be searched.

5.5.2 Setting Code Line Breakpoints

A second, more flexible way of setting breakpoints, is by adding code line breakpoints. In GOAL, breakpoints can be associated with four constructs in an agent program: a module, the condition of an action rule, the action part of an action rule, and an action specification. Breakpoints can be added to these constructs by opening a MAS project file in Eclipse, and adding breakpoints at *the first line associated with the construct*. That is, put a breakpoint next to the line with the module name, the first line associated with an action rule, or the the line with the action name. You can do this by double-clicking in the left border of the editor.

GOAL will automatically allocate placed breakpoints to their appropriate position when the MAS is started by attaching any placed breakpoint to the first suitable location after its current position.

There are two types of breakpoints: the ALWAYS and the CONDITIONAL type. A CONDITIONAL breakpoint shows as an orange icon in the left border. It can only attach to the action part of an action rule. The ALWAYS type shows as a red icon in the left border, and attaches to the other three types. An ALWAYS breakpoint can be changed into a CONDITIONAL breakpoint by double-clicking on it again. Double-clicking on a CONDITIONAL breakpoint will remove the breakpoint. An overview of breakpoints that have been set can be seen in the Debug Perspective by opening the Breakpoints tab in the top left window. Note that explicit breakpoints that have been added to a file are not permanently saved: upon closing Eclipse, all breakpoints set are lost. This is an easy way to clear all breakpoints.

An example of two breakpoints that were added to a file is illustrated in Figure 5.2.

stackBuilder.goal

```
        % moves a block on top of another block or to the table.
        move(X, Y) {
           % a block can only be moved elsewhere if that block and the place to move it to are both clear.
           % on(X, Z) retrieves the current (and soon to be old) position where block X sits on.
           % not( on(X,Y) ) prevents an agent from moving a block on the table to another place on the table.
           % because a block cannot be put on top of itself not( X=Y ) is included in the precondition.
           pre { clear(X), clear(Y), on(X, Z), not( on(X, Y) ), not( X=Y ) }
           % effect of moving block X on top of Y is that we have on(X, Y).
           % after moving, block X is no longer on top of Z and we remove on(X, Z).
           post { not( on(X, Z) ), on(X, Y) }
        }
    }
}

% Decide on an action to perform in Blocks World.
main module [exit=nogoals] {
    program {
        #define misplaced(X) a-goal(tower([X| T])).
        #define constructiveMove(X,Y) a-goal(tower([X, Y| T])), bel(tower([Y| T])).

        if constructiveMove(X, Y) then move(X, Y).
        if misplaced(X) then move(X, table).
    }
}
```

Figure 5.2: An ALWAYS and a CONDITIONAL Breakpoint added to the `BlocksWorld` example

Different from the built-in breakpoints, even if a multi-agent system is put in run mode, the system will halt on any explicit breakpoints that have been set.

5.6 Prolog Level Debugging

GOAL does not support tracing of Prolog queries inside the Prolog inference engine. If you need this level of detail while you are debugging an agent, it is useful to be able to export the contents of a mental state to a separate file that can be imported into Prolog. Then use the Prolog tracer to analyze the code.

5.6.1 Prolog Exceptions

When an exception occurs in the Prolog engine (SWI Prolog), error messages may get a bit hard to decipher. This section explains how to read such messages.

A typical PrologException will show up typically like this:

```
WARNING:  jpl.PrologException: PrologException:
          error(instantiation_error, context(:(system, /(>=, 2)), _1425))
query=true,beliefbase6: (openRequestOnRoute(L,D,T))
```

The first of the two lines indicates the message that Prolog gave us. The second line shows the query that caused the warning. To read the first line, notice the part `error(CAUSE,DETAILS)` inside it.

The CAUSE part gives the reason for the failure, the DETAILS gives the context of the error. Several types of CAUSE can occur, amongst others:

1. type_error(X,Y): an object of type X was expected but Y was found. For instance, in the context of arithmetic X can be 'evaluable' and Y a non-arithmic function.

2. instantiation_error: the arguments for an operator are insufficiently instantiated. For example, if we ask X>=8 with X a free variable, we will get an error like the example above. The context in such a case would be `context(:(system, /(>=, 2)), _1425)`. The first part,

`system:>=/2` refers to the >= operator (which has arity 2 and is part of the system module) and the _1425 refers to the free variable (which apparently was even anonymous hence is not of much help). Note that these messages commonly contain pure functional notation where you might be more used to the infix notation. For instance `system:X` appears as `:(system,X)` and >=/2 appears as /(>=,2).

3. representation_error: the form of the predicate was not appropriate, for instance a bound variable was found while an unbound variable was expected

4. permission error: you have insufficient permissions to execute the predicate. Typically this occurs if you try to redefine system predicates.

5. resource_error: there are not enough resources of the given type.

5.7 Runtime Preference Settings

Most settings can be modified from the preferences in Eclipse. Access them through opening the GOAL category in Window → Preferences →, and selecting the Runtime page. This section discusses those settings.

The 'Other' section offers the option to always run the middleware locally. This checkbox is by default disabled. If you turn on this checkbox, the default localhost will be used as middleware host.

The 'Default KR Language' selection allows you to choose between the default swiprolog implementation and any alternative language. Currently, only swiprolog is actively supported.

In the 'High performance' section, selecting the 'sleep agents when they repeat same action' enables GOAL to skip agents that seem to be idle, which relocates CPU time to other agents that really are working on something. GOAL guesses that an agent is idle if the agent is receiving the same percepts and mails as last step on the input, and picks the same actions as output. The sleep is canceled when the percepts/mails of the agent change. This setting is on by default.

IMPORTANT: with 'sleep agents' selected, your agent may go to sleep when it is repeating actions. This may be counter-intuitive, and may cause GOAL programs to malfunction. In doubt, you should turn off this option.

To remove agents automatically after they have been killed, select the 'Remove agents when they are killed' option. This option is useful in environments where large numbers of agents appear and disappear while GOAL is running.

Finally, in the 'learning' section, agent learning can be turned on. Optionally, a global learned-behavior file can be set here as well.

Chapter 6

Known Issues and Workarounds

Below we have listed some known issues and possible workarounds when available.

- **Nothing happens when pressing the Step and Run buttons**.
  There are a number of possible explanations for this you can check.

  1. First, make sure that the environment is ready to run. For example, when using the Elevator environment the environment will respond to GOAL only if you selected *GOAL Controller* available in this environment.
  2. You may be stepping an agent that is suspended somehow, e.g. the environment may be refusing to serve that agent, or it may be the turn of another agent.

- **The environment does not work as expected**.
  Check the documentation that comes with the environment. Most environments need to be set up before the agents can connect and perform actions.

- **The Wumpus environment hangs when the agent shoots an arrow**.
  The Wumpus World always has to be enclosed by means of walls. If an agent shoots an arrow into open space, the arrow will continue forever, causing the system to hang.

- **I do not see an environment after launching a MAS**.
  The environment may be hidden behind another window. This is due to a known bug in Java for Windows. You need to install Java 7 or higher to fix this issue. Java 7 is available from http://download.java.net/jdk7/binaries/.

For problems specific with the Eclipse interface, these two steps can be taken:

- Reset the GOAL and/or GOAL Debug perspective(s) by selecting 'Window → Reset Perspective' when the perspective is opened.

- Restore the default GOAL preferences through 'Window → Preferences → GOAL → Logging → Restore Defaults → Apply', and do the same for the 'Runtime' category.

6.1 Reporting Issues

If you discover anything you believe to be an error or have any other issues you run into, please report the error or issue to goal@ii.tudelft.nl. Your help is appreciated!

To facilitate a quick resolution and to help us reproduce and locate the problem, please include the following information in your error report:

1. A short *description* of the problem.
2. All relevant *source files* of the MAS project.
3. Relevant warning or error *messages from the Feedback area*, especially from the Console and/or Parse Info tab. Please also include *Java details and stack trace information* if available (see Section 5.4 for adjusting settings to obtain this information).
4. If possible, a *screen shot* of the situation you ran into.

Bibliography

[1] Tristan Behrens. Environment interface standard project. http://sourceforge.net/projects/apleis/.

[2] Tristan Behrens, Koen V. Hindriks, and Jrgen Dix. Towards an environment interface standard for agent platforms. *Annals of Mathematics and Artificial Intelligence*, pages 1–35, 2010. 10.1007/s10472-010-9215-9.

[3] Koen V. Hindriks. *GOAL Programming Guide*. Delft University of Technology, 2014. http://ii.tudelft.nl/trac/goal/wiki/WikiStart#Documentation.

[4] Koen V. Hindriks. The Goal Agent Programming Language. http://ii.tudelft.nl/trac/goal, 2014.